# Linked List

Great video about how to implement linked list in C:

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFiudyP6wSXmykrn67Ykqib>

## Definition

## Operations & Complexities

## Applications

## Linked List vs Array

Both arrays and linked list can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.





**Drawbacks of arrays:**

1. The size of the arrays is fixed: We must know the upper limit on the number of elements in advance. Also, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

2. Inserting a new element to an array is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, ...].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

3. Deletion is also expensive with arrays until unless some special techniques are used.

For example, to delete 1010 in id[], everything after 1010 has to be moved.

**Linked list provides following two advantages over arrays:**

1. Dynamic size

2. Ease of insertion/deletion

**But linked lists have following drawbacks:**

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So, we cannot do binary search with linked lists.

2. Extra memory space for a pointer is required with each element of the list.

3. Arrays have better cache locality that can make a pretty big difference in performance.

## Why double pointers are used in linked list?

It is clear that both methods (single pointer and double pointer) lead to the **same result**. The only difference is **what will be changed afterward**.

Double pointers are used as **arguments** of function when the function modifies and updates the linked list without needing to return the value (address or data) of the list again.

When using single pointers as arguments of function that modifiers and updates the linked list, we must return the value (address or data) of the list. Or else, the effect won’t be noticed.

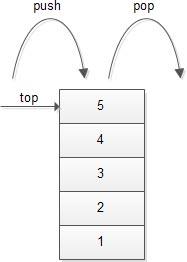
Briefly, remember the simple C rule: If you want to **modify local variable of one function inside another function**, pass pointer to that variable. It is called "call by pointers". In this case, the pointer is C’s way of implementing "call by reference" when there is no reference variable.

For example, you want to add a new node before the head (first node) of the list, and hence, the pointer pointing to the first node will be then changed. When you exit this function, you want this change to reflect in the calling function and the following code in the main() function (suppose you call this function in the main()). In this case, you have to use a double pointer. One of them is to indicate that you are passing an address and another is to make the changes available to the calling function (to achieve call by reference).

# Stack

## Definition

Stack is a linear data structure that allows adding and removing elements in a specific order. In particular, every time an element is added, it goes on the top of the stack. The only element that can be removed is the one at the top of the stack. In other words, **the first item added to a stack will be the last item removed from it**. As a result, a stack is said to have "last in first out" behavior (or *LIFO*).



A typical example of using stack is function calling. A function calls another function, which in turn calls a third function; it's important that the third function return back to the second function rather than the first one.

*You might not know!*

The "call stack" is the term used for the list of functions either executing or waiting for other functions to return.

## Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Meaning** | **Time Complexity** | **Space Complexity** |
| S.push(val) | Adds an item in a stack. If the stack is full, it is said to be an *Overflow* condition | O(1) | O(1) |
| S.pop() | Removes an item from a stack. If the stack is empty, it is said to be an *Underflow* condition | O(1) | O(1) |
| S.top() or S.peek() | Returns a reference to the top most element of a stack | O(1) | O(1) |
| S.empty() | Returns true if stack is empty, else false | O(1) | O(1) |
| S.full() | Returns true if stack is full, else false | O(1) | O(1) |
| S.size() | Returns the size of a stack | O(1) | O(1) |

## Applications

* [Balancing of symbols](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/)
* [Infix to Postfix](http://quiz.geeksforgeeks.org/stack-set-2-infix-to-postfix/) /Prefix conversion
* Redo-undo features at many places like editors, photoshop
* Forward and backward feature in web browsers
* Used in many algorithms like [Tower of Hanoi,](https://www.geeksforgeeks.org/recursive-functions/)[tree traversals](https://www.geeksforgeeks.org/618/), [stock span problem](https://www.geeksforgeeks.org/the-stock-span-problem/), [histogram problem](https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/).
* Other applications can be Backtracking, [Knight tour problem](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/), [rat in a maze](https://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/), [N queen problem](https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/) and [sudoku solver](https://www.geeksforgeeks.org/backtracking-set-7-suduku/)
* In Graph Algorithms like [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/) and [Strongly Connected Components](https://www.geeksforgeeks.org/strongly-connected-components/)

## Implementation

There are some ways to implement a stack:

* Using array
* Using linked list
* Using vector

**Example 1**: Using array (🡪 static stack)

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 1000

class Stack

{

private:

    int top;                    // Index of the top item

public:

    int arr[MAX\_SIZE];          // Maximum size of Stack

    Stack() {

        top = -1;

    }

    bool push(int x) {

        if (isFull()) {

            cout << "Stack Overflow";

            return false;

        }

        else {

            arr[++top] = x;

            cout << x << " pushed into stack\n";

            return true;

        }

    }

    int pop() {

        if (top < 0) {

            cout << "Stack Underflow";

            return 0;

        }

        else {

            int x = arr[top--];

            return x;

        }

    }

    int peek() {

        if (isEmpty()) {

            cout << "Stack is Empty";

            return 0;

        }

        else {

            int x = arr[top];

            return x;

        }

    }

    bool isEmpty() { return (top < 0); }

    bool isFull() { return (top >= (MAX\_SIZE - 1)); }

};

int main()

{

    class Stack s;

    s.push(10);

    s.push(20);

    s.push(30);

    // print top element of stack after popping

    cout << "Top element is: " << s.peek() << endl;

    // print all elements in stack:

    cout << "Elements present in stack: ";

    while(!s.isEmpty()) {

        cout << s.peek() <<" ";

        s.pop();

    }

    return 0;

}

**Example 2**: Using linked list (🡪 dynamic stack)

#include <iostream>

class Node {

public:

    int data;

    Node\* next;

    Node(int value) {

        data = value;

        next = nullptr;

    }

};

class Stack {

private:

    Node\* top;

public:

    Stack() {

        top = nullptr;

    }

    void push(int element) {

        Node\* newNode = new Node(element);

        newNode->next = top;

        top = newNode;

    }

    int pop() {

        if (isEmpty()) {

            std::cout << "Stack underflow!" << std::endl;

            return -1; // Error value

        }

        Node\* temp = top;

        int topElement = top->data;

        top = top->next;

        delete temp;

        return topElement;

    }

    int peek() const {

        if (isEmpty()) {

            std::cout << "Stack is empty!" << std::endl;

            return -1; // Error value

        }

        return top->data;

    }

    bool isEmpty() const {

        return top == nullptr;

    }

    int size() const {

        int count = 0;

        Node\* current = top;

        while (current != nullptr) {

            count++;

            current = current->next;

        }

        return count;

    }

};

int main() {

    Stack stack;

    stack.push(10);

    stack.push(20);

    stack.push(30);

    std::cout << "Stack size: " << stack.size() << std::endl;

    while (!stack.isEmpty()) {

        std::cout << "Top element: " << stack.peek() << std::endl;

        stack.pop();

    }

    return 0;

}

**Example 3**: Using vector (🡪 dynamic stack)

#include <iostream>

#include <vector>

class Stack {

private:

    std::vector<int> data;

public:

    void push(const int& element) {

        data.push\_back(element);

    }

    int pop() {

        if (isEmpty()) {

            throw std::runtime\_error("Stack is empty");

        }

        int topElement = data.back();

        data.pop\_back();

        return topElement;

    }

    int top() const {

        if (isEmpty()) {

            throw std::runtime\_error("Stack is empty");

        }

        return data.back();

    }

    bool isEmpty() const {

        return data.empty();

    }

    size\_t size() const {

        return data.size();

    }

};

int main() {

    Stack stack;

    stack.push(10);

    stack.push(20);

    stack.push(30);

    std::cout << "Stack size: " << stack.size() << std::endl;

    while (!stack.isEmpty()) {

        std::cout << "Top element: " << stack.top() << std::endl;

        stack.pop();

    }

    return 0;

}

# Queue

## Definition

Queue is a linear data structure that allows adding and removing elements in a specific order. To understand a queue, think of a cafeteria line: new people are added to the line at the back; the first person in line is served first, and the last person is served last. So, **in a queue the first item added to it will be the first item removed from it**. As a result, a queue is said to have "first in first out" behavior (or *FIFO*). That is opposite to a [stack](#_2et92p0).



*Note:*

Although the concept is simple, programming a queue is not as simple as programming a *stack*.

Let's go back to the example of the cafeteria line. Let's say one person leaves the line. Then what? Everyone in line must step forward, right? Now, imagine if only one person could move at a time. So, the second person steps forward to fill the space left by the first person, and then the third person steps forwards to fill the space left by the second person, and so on. Now imagine that no one can leave or be added to the line until everyone has stepped forward. You can see the line will move very slowly.

It is not difficult to program a queue that works, but it is **quite touch to make a queue that works fast**!

## Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Meaning | Time Complexity | Space Complexity |
| Q.enqueue(val) | Adds an item to a queue. If the queue is full, it is said to be an *Overflow* condition | O(1) | O(1) |
| Q.dequeue() | Removes an item from a queue. If the queue is empty, it is said to be an *Underflow* condition | O(1) | O(1) |
| Q.front() | Get the front item from a queue | O(1) | O(1) |
| Q.back() or Q.rear() | Get the last item from a queue | O(1) | O(1) |
| Q.empty() | Returns true if queue is empty, else false | O(1) | O(1) |
| Q.full() | Returns true if queue is full, else false | O(1) | O(1) |
| Q.size() | Get the numer of elements in a queue | O(1) | O(1) |

## Applications

Queue is used when things don’t have to be processed immediately, but have to be processed in FIFO order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property makes queue useful in following scenarios.

* When a resource is shared among multiple consumers. For examples, CPU scheduling or disk scheduling.
* When data is transferred asynchronously (data isn’t necessarily received at same rate as sent) between two processes. For examples, IO buffers, pipes, file IO, sockets, etc.
* Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario.

## Implementation

There are some ways to implement a queue:

**Using array**:

* The first method is to make an array and shift all the elements to accommodate enqueues and dequeues. This is slow, because with many elements, the shifting takes time.
* The second method is, instead of shifting the elements, shifting the enqueue and dequeue points. Imagine that cafeteria line again. If the front of the line continually moves backward as each person leaves the line, then people don't need to step forward or backward, which saves time.???
* This method is much more complicated than the first one. Instead of keeping track of just the enqueue point (the "end"), we also need to keep track of the dequeue point (the "front"). This all gets even more complicated when we realize that after a bunch of enqueues and dequeues, the line will need to wrap around the end of the array. Think of the cafeteria line. As people enter and leave the line, the line moves farther and farther backwards, and eventually it will circle the entire cafeteria and end up at its original location.???

<https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/>

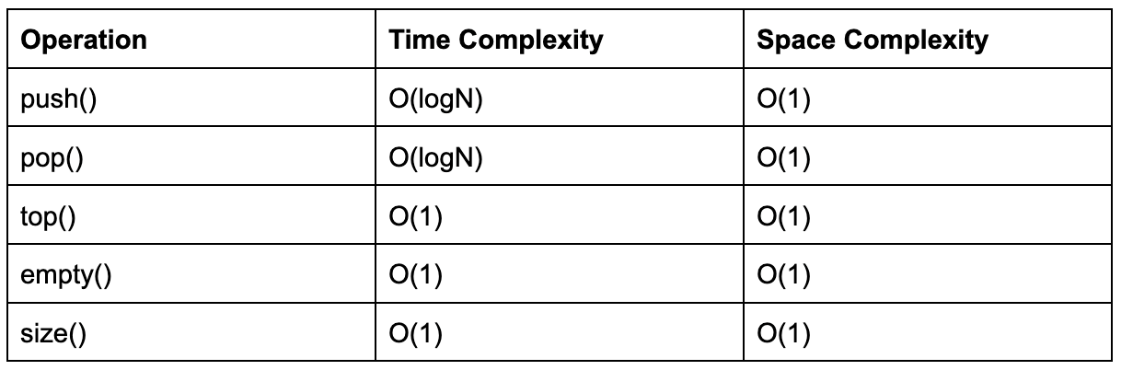
**Using linked list:**

Example:

* <https://mathcs.clarku.edu/~fgreen/courses/cs170/FunctionPointers/Queue.c> (perfect, no memory leak)
* <https://gist.github.com/ArnonEilat/4471278> (can cause memory leak)

## Different Types of Queues

### Priority Queue

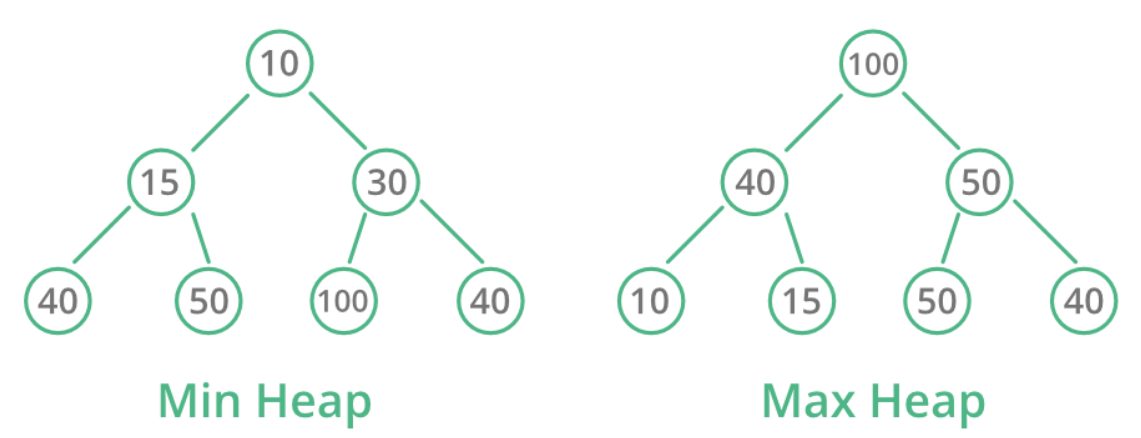


### Circular Queue

# Heap

## Definition

A heap is a special **tree-based data structure** in which the tree is a complete [Binary Tree](#_Binary_Tree).



Types:

* **Max-Heap**: The **key present at the root node must be greatest** among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.
* **Min-Heap**: The **key present at the root node must be smallest** among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

Other types:

1. [Binomial Heap](https://www.geeksforgeeks.org/binomial-heap-2/)
2. [Fibonacci Heap](https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/)
3. [Leftist Heap](https://www.geeksforgeeks.org/leftist-tree-leftist-heap/)
4. [K-ary Heap](https://www.geeksforgeeks.org/k-ary-heap/)

## Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Meaning | Time Complexity | Space Complexity |
| Heapify | Create a heap from an array. |  |  |
| Insertion | Insert an element in existing heap. | O(log N) |  |
| Deletion | Delete the top element of the heap or the highest priority element, and then organizing the heap and returning the element. | O(log N) |  |
| Peek | Find the top element of the heap. |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

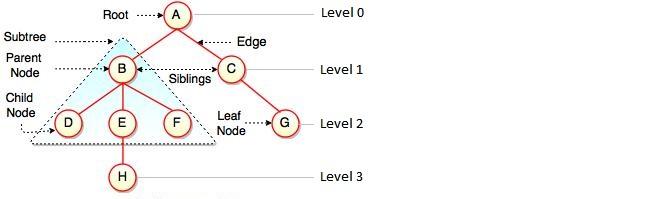
## Applications

# Tree

## General Tree

### Definition

Tree is a non-linear data structure which represents the nodes connected by edges. It’s used to store the information in the form of **hierarchy style**.



Terms:

* *Path* – Represents the sequence of nodes along the edges of a tree.
* *Visiting* – Represents checking the value of a node when control is on the node.
* *Traversing* – Represents passing through nodes in a specific order.
* *Keys* − Represents a value of a node based on which a search operation is to be carried out for a node.
* *Height of Tree* – Represents the height of its root node.
* *Height of Node* – Represents the number of edges on the longest path between that node and a leaf.
* *Depth of Node* – Represents the number of edges from the tree's root node to the node.
* *Degree of Node* – Represents a number of children of a node.

### Traversal

In order to process trees, we need a mechanism for traversing them. Each node is processed only once but it may be visited more than once. As we have already seen **in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order. But, in tree structures there are many different ways**.

Starting at the root of a BT, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (denoted with "D"), traversing to the left child node (denoted with "L"), and traversing to the right child node (denoted with "R"). This process can be easily described through recursion.

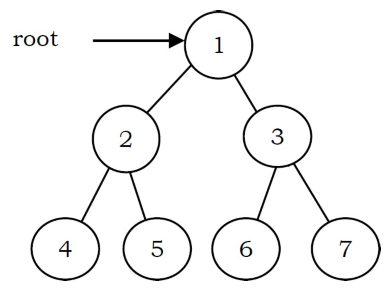
*So, we can classify the traversals into several styles:*

* *In-order traversal (LDR)*
* *Pre-order traversal (DLR)*
* *Post-order traversal (LRD)*

*There is another traversal method which does not depend on the above orders. It is:*

* *Level-order traversal*

Let us use the diagram below for the remaining discussion.



#### In-Order Traversal (LDR)

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

In in-order traversal, the root is visited between the subtrees. It is defined as follows:

1. Traverse the left subtree in in-order.
2. Visit the root.
3. Traverse the right subtree in in-order.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

#### Pre-Order Traversal (DLR)

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

In pre-order traversal, each node is processed before (pre-) either of its subtrees. It is defined as follows:

1. Visit the root.
2. Traverse the left subtree in pre-order.
3. Traverse the right subtree in pre-order.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

#### Post-Order Traversal (LRD)

The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1

In post-order traversal, the root is visited after both subtrees. It is defined as follows:

1. Traverse the left subtree in post-order.
2. Traverse the right subtree in post-order.
3. Visit the root.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

#### Level-Order Traversal

The nodes of the tree are visited in the order: 1 2 3 4 5 6 7

Level-order traversal is defined as follows:

1. Visit the root (level 0).
2. Visit all nodes at level 1, from left to right.
3. Go to the next level and visit all the nodes at that level.
4. Repeat this until all levels are completed.

*Time Complexity: O(n).*

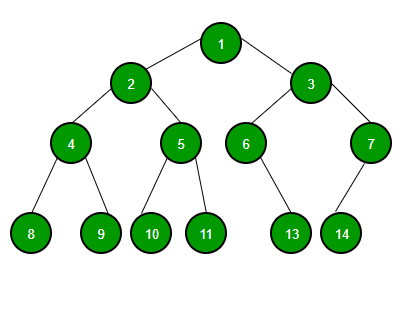
*Space Complexity: O(n). Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.*

## Binary Tree

### Definition

BT is a special tree used for data storage purposes. It has a special condition that **each node can have a maximum of two children**.

A BT has the benefits of both an ordered array and a linked list – **Search in BT is as quick as in a sorted array, and insertion or deletion in BT are as fast as in linked list**.



### Operations & Complexities

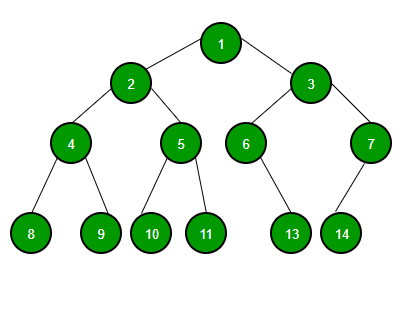
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(n) | O(n) |
| **Insert** |  | O(n) | O(n) |
| **Delete** |  | O(n) | O(n) |

Where: n is number of nodes

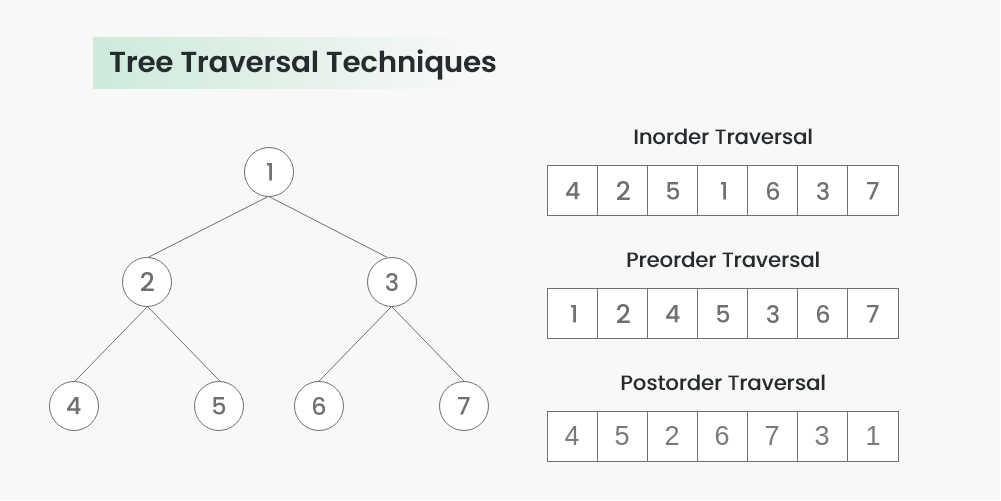
### Definition

BT is a special tree used for data storage purposes. It has a special condition that **each node can have a maximum of two children**.

A BT has the benefits of both an ordered array and a linked list – **Search in BT is as quick as in a sorted array, and insertion or deletion in BT are as fast as in linked list**.



### Traversal



#### In-Order Traversal (LDR)

Used algorithm: Depth First Search (DFS)

|  |  |
| --- | --- |
|  | 4  2  5  1  6  3  7 |

#### Pre-Order Traversal (DLR)

Used algorithm: Depth First Search (DFS)

1 2 4 5 3 6 7

#### Post-Order Traversal (LRD)

Used algorithm: Depth First Search (DFS)

4 5 2 6 7 3 1

#### Level-Order Traversal

Used algorithm: Breadth First Search (BFS)

<https://www.geeksforgeeks.org/level-order-tree-traversal/>

### Implementation

#### In-Order Traversal (LDR)

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node\* left, \* right;

};

Node\* newNode(int data) {

Node\* temp = new Node;

temp->data = data;

temp->left = NULL;

temp->right = NULL;

return temp;

}

void printInorder(struct Node\* node) {

if (node == NULL) {

return;

}

// First recur on left child

printInorder(node->left);

// Then print the data of node

cout << node->data << " ";

// Now recur on right child

printInorder(node->right);

}

// Driver code

int main() {

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

// Function call

cout << "Inorder traversal of binary tree is \n";

printInorder(root);

return 0;

}

Output:

4 2 5 1 6 3 7

#### Pre-Order Traversal (DLR)

...

void printInorder(struct Node\* node) {

if (node == NULL) {

return;

}

cout << node->data << " ";

printInorder(node->left);

printInorder(node->right);

}

...

Output:

1 2 4 5 3 6 7

#### Post-Order Traversal (LRD)

...

void printInorder(struct Node\* node) {

if (node == NULL) {

return;

}

printInorder(node->left);

printInorder(node->right);

cout << node->data << " ";

}

...

Output:

4 5 2 6 7 3 1

#### Level-Order Traversal

### Applications

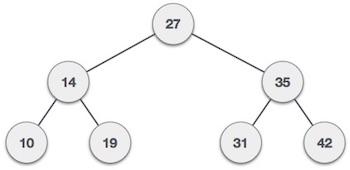
Following is the some of the applications where BT plays an important role:

* Expression trees used in compilers.
* Huffman coding trees used in data compression algorithms.
* [Binary Search Tree](#_1fob9te) (BST), which supports search, insertion and deletion on a collection of items in O(logn) (average).
* [Priority Queue](#_gjdgxs) (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (worst case).

## Binary Search Tree

### Definition

BST exhibits a special behavior of a binary tree. **A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value**. Also, equal node values are not allowed in BST.



### Operations & Complexities

Complexity for all BST operations depends on BST height (h). It's like:

|  |  |
| --- | --- |
| Worst case | Best case |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** | - Start at root  - Take left and right children as necessary | O(logh) | O(h) |
| **Insert** | - Similar to search, but look for null pointer | O(logh) | O(h) |
| **Delete** | - Search (if necessary)  - Different cases based on tree degree (number of children):  + 0 (meaning it’s a leaf node): just delete  + 1: similar to doubly linked list  + 2: swap with max (left) and delete (0 or 1 children)  - Update size and root | O(logh) | O(h) |

#### Tree Rotations

* Based around a pivot node (\*)
* Two directions: left and right

|  |  |
| --- | --- |
| * Left rotation: * Parent becomes left child * Pivot becomes parent * Old left child becomes parent right child * Pivot is new "root" | * Right rotation:   + Parent becomes right child   + Pivot becomes parent   + Old right child becomes parent left child   + Pivot is new "root" |

#### Insertion

#### Deletion

### Traversal

### Implementation

### Applications

## Balanced Binary Search Tree

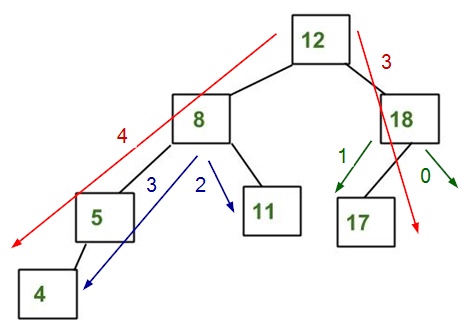
Balanced BST is a BST with self-balancing capability. It is divided into following subtypes:

### AVL Tree

#### Definition

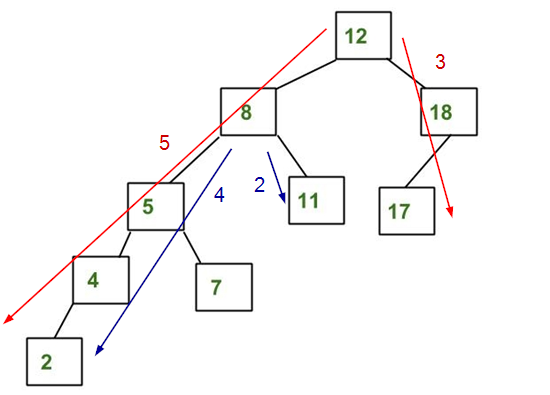
AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing BST where the **difference between heights of left and right subtrees cannot be more than one for all nodes**. The balance is maintained using *tree rotations* as nodes inserted and deleted.

Example: AVL Tree



It is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

Example: NOT AVL Tree



It is not AVL because differences between heights of left and right subtrees for 8 is 4 (greater than 1) and for 12 is 5 (greater than 1).

#### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

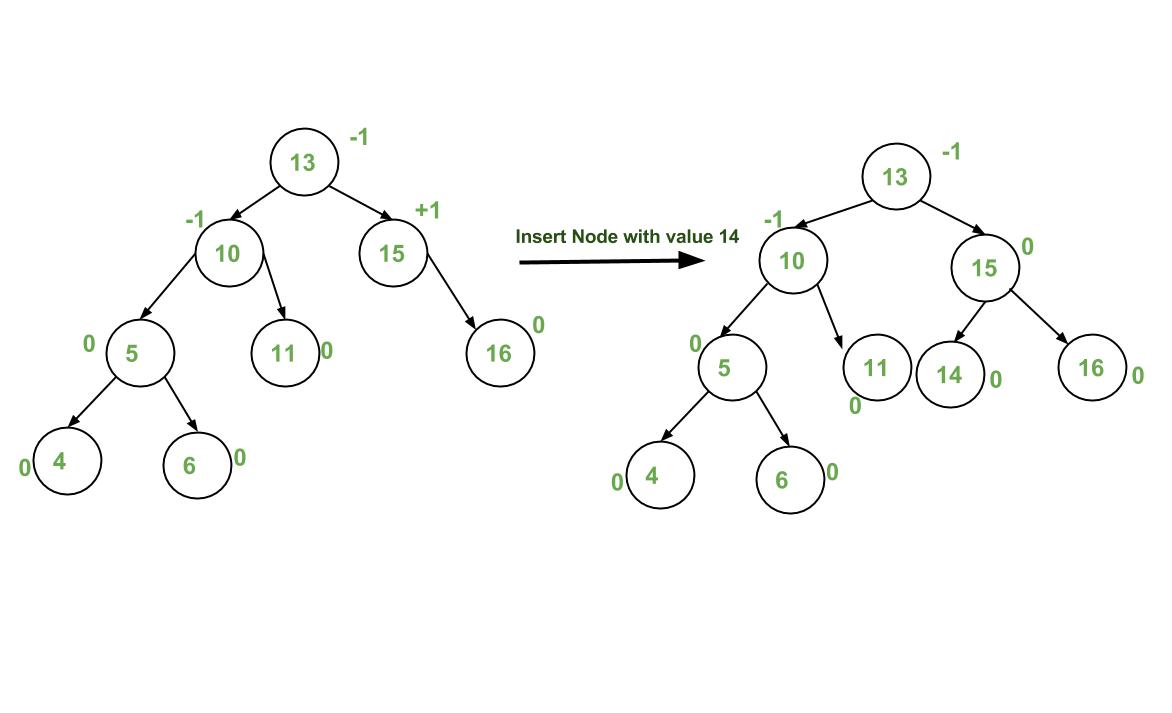
*More details:*

Most of the BST operations take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed binary tree. If we **make sure that height of the tree remains O(logn)** after every insertion and deletion, then we can guarantee an upper bound of O(logn) for all these operations. The height of an AVL tree is always O(logn) where n is the number of nodes in the tree.

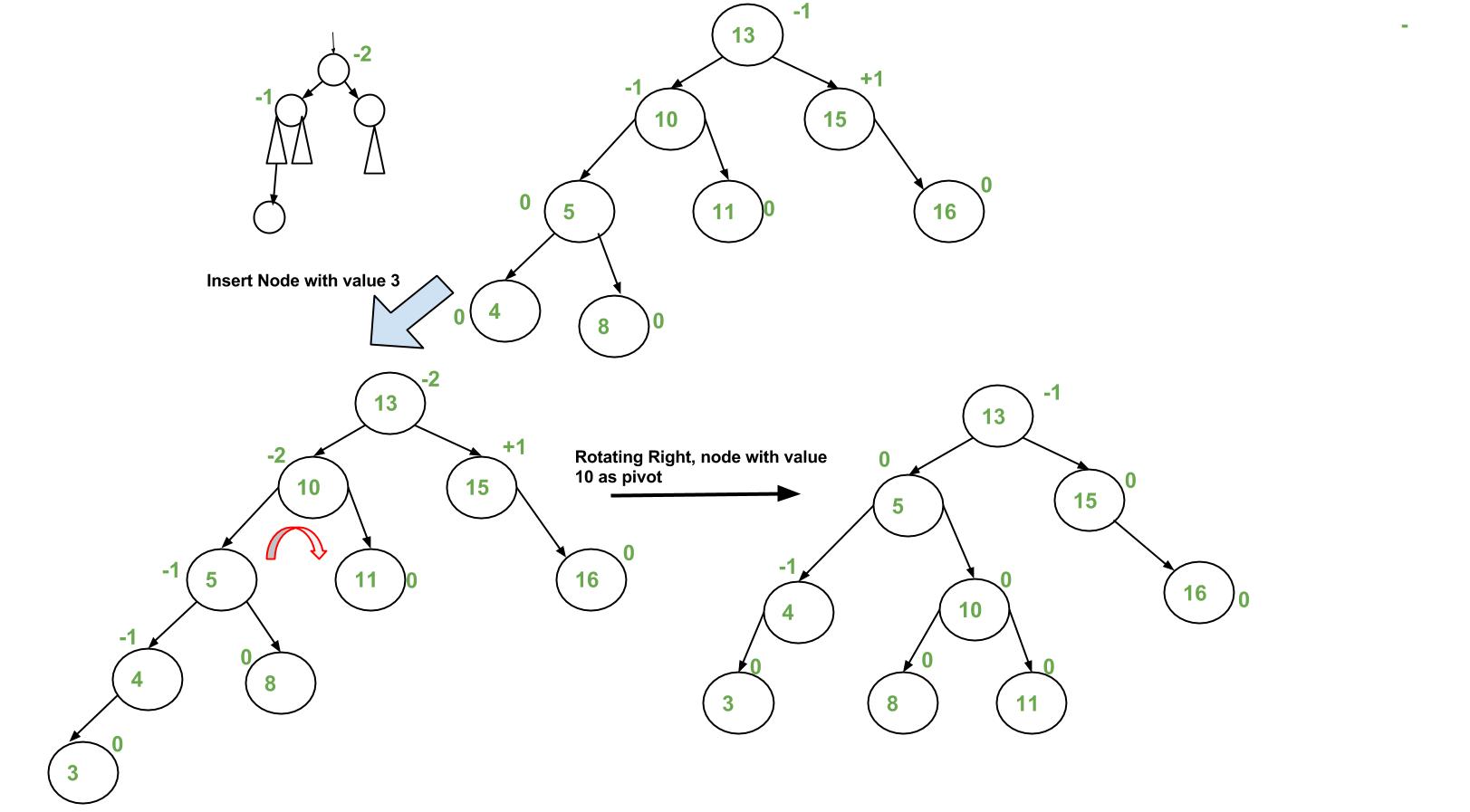
##### Insertion

Every time a new node is inserted, the tree must be re-balanced (if needed). One of following five cases can happen:

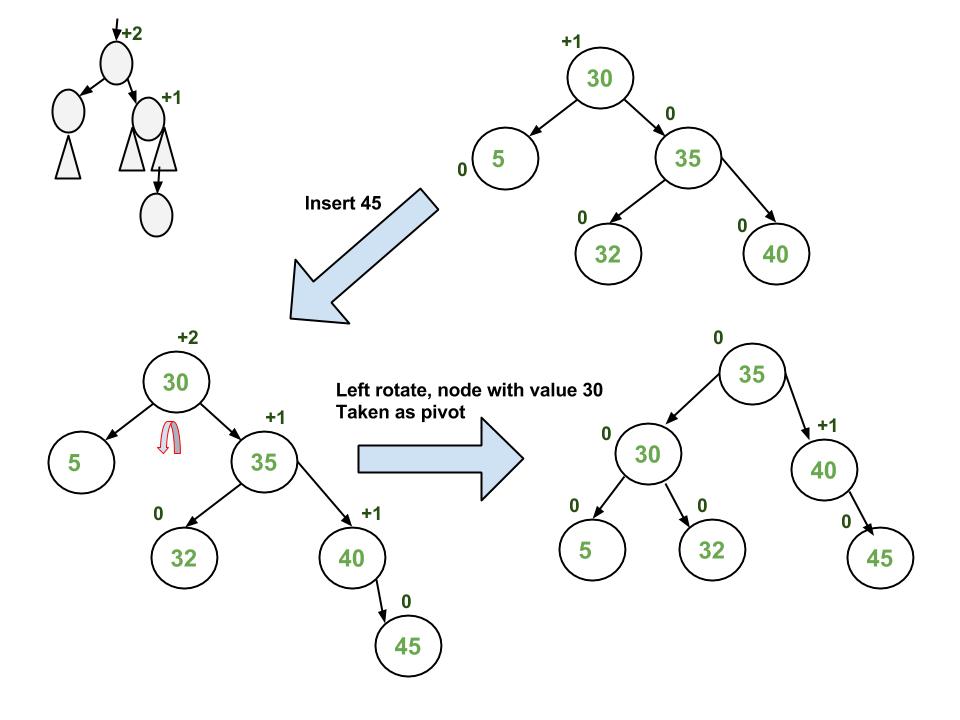
Case 1: No rotation needed



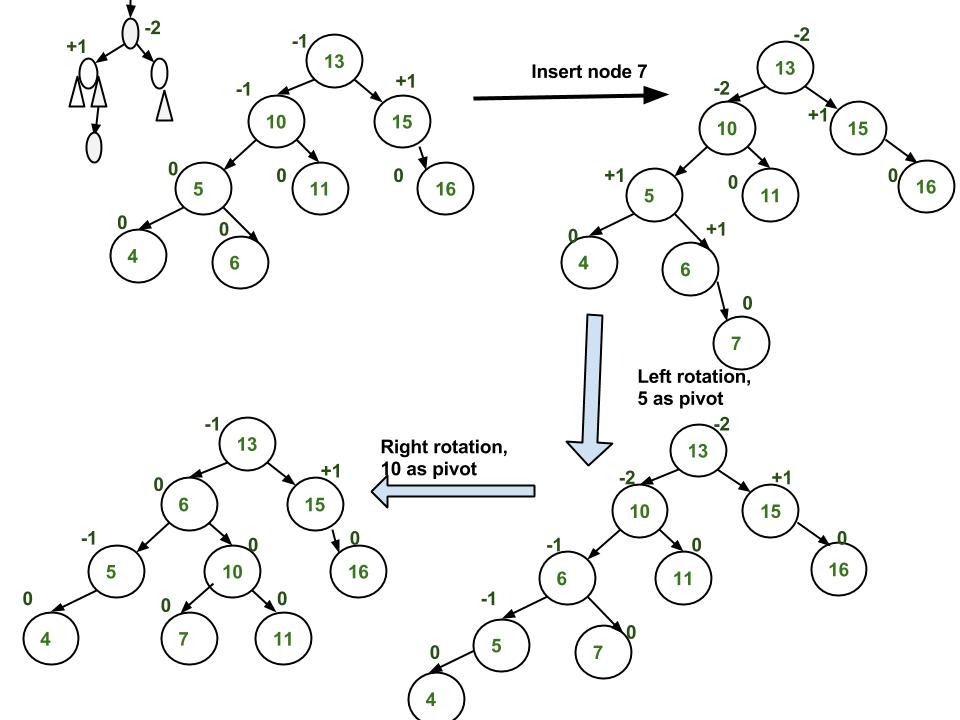
 Case 2: Right rotation



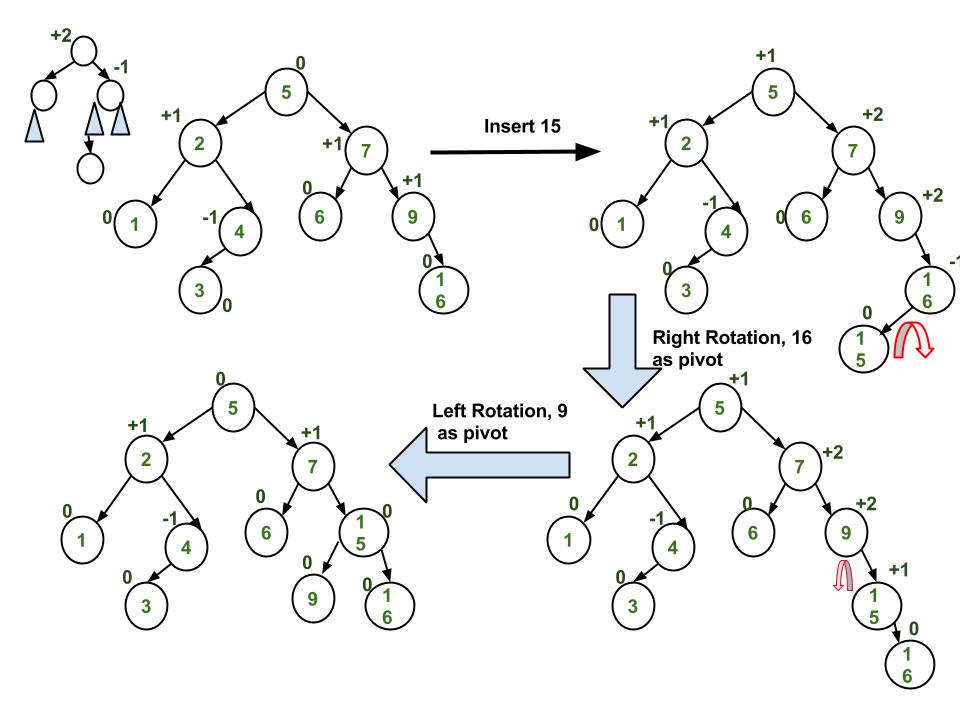
Case 3: Left rotation



Case 4: Left rotation, then right rotation



Case 5: Right rotation, then left rotation



Implementation: <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

##### Deletion

Every time a node is deleted, the tree must be re-balanced (if needed). And there are five cases can happen, just similar to insertion.

Implementation: <https://www.geeksforgeeks.org/avl-tree-set-2-deletion/>

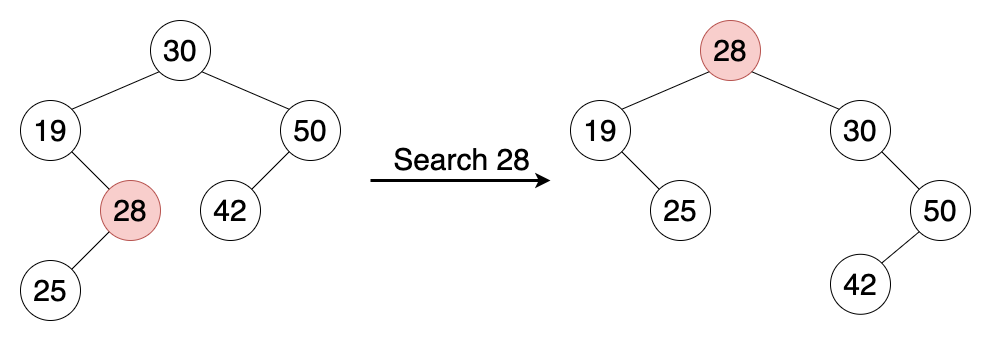
#### Applications

* Used in situations where frequent insertions are involved.
* Used in Memory management subsystem of the Linux kernel to search memory regions of processes during preemption.

### Splay Tree

#### Definition

A splay tree is a self-balancing BST. After performing a search, insertion or deletion, splay trees perform an action called *splaying* where the tree is rearranged (using rotations) so that the **particular element is placed at the root of the tree**.



#### Basic Operations & Time Complexities

#### Applications

* Used to implement caches
* Used in garbage collectors.
* Used in data compression

### Red-Black Tree

#### Definition

#### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

#### Applications

* As a base for data structures used in computational geometry.
* Used in the Completely Fair Scheduler used in current Linux kernels.
* Used in the epoll system call implementation of Linux kernel.

### Comparions

**AVL Tree vs. Red Black Tree**

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So, if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then AVL tree should be preferred over Red Black Tree.

**AVL Tree vs. Splay Tree**

Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node. However, unlike AVL tree, a splay tree can change even with read-only operations like search.

## Balanced Search Tree

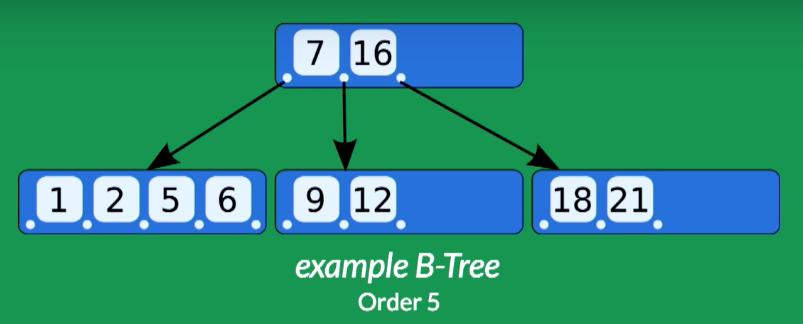
### B-Tree

#### Definition

Should watch it first: <https://www.youtube.com/watch?v=C_q5ccN84C8&ab_channel=FullstackAcademy>

B-Tree is a self-balancing search tree, which contains **multiple nodes storing data in sorted order**. Each node can have **multiple children** and consists of **multiple keys**.

Example:



**Root node**

**Child node**

**Key**

The above B-Tree has:

* 1 root containing 2 keys: 7 and 16
* 3 child nodes. The left most node containing keys whose values are < 7. The middle node containing keys whose values are > 7 and < 16. The right most node containing keys whose values are > 16.
* Order of 5 (because its node – the left most one – has at most 5 children)

Properties:

A B-Tree of order **m** has:

* All leaves appear in the same level.
* Every node has at most **m** children.
* A non-leaf node with **k** children contains **k-1** keys.
* The root has at least two children if it is not a left node.
* Every non-left node (except the root) has at least a **ceiling of m/2** children.

#### Basic Operations & Time Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

**Insertion: 7:30**

#### Applications

* Used in database indexing to speed up the search.
* Used in file systems to implement directories.

*You might not know!*

To understand the use of B-Trees, we must think of the huge amount of data which cannot fit in main memory. So, we have to put the data in disk. But accessing and reading data from disk takes much more time than from main memory.

In this case, we can take advantage of B-Trees to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, etc.) require O(h) disk accesses where h is the height of the tree. But the height of B-Trees is kept low by putting maximum possible keys in a B-Tree node, so total disk accesses for most of the operations are reduced significantly compared to balanced BST (like AVL Tree, Red-Black Tree, etc.).

To conclude, **whenever you deal with some kind of external memory and the time to access the data of a node greatly exceeds the time spent processing that data (such as big databases)**, consider using B-Trees.

# Graph

## Definition

A graph is a non-linear data structure **consisting of vertices (nodes) and edges** which are lines/arcs connecting any pair of nodes.

Mathematically, a graph G = (V, E) is a set of vertices V and edges E where each edge (u, v) is a connection between vertices: u, v ∈ V.

For example:



The above graph has a set of vertices V = {0, 1, 2, 3, 4} and a set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Terms:

* **Neighbors**: Two vertices u and v are neighbors if they’re a edge (u, v) connecting between them.

E.g.: 0 and 1 are neighbors, 0 and 4 are neighbors 🡪 neighbors(0) = {1, 4}

* **Degree**: Degree(v) is the number of edges connected to v.

E.g.: degree(1) = 3, degree(2) = 2

* **Path**: Sequence of vertices connected by edges.

E.g.: 0 -> 1 -> 2 -> 3 is a path

* **Path length***:* Number of edges in a path.

E.g.: 0 -> 1 -> 2 -> 3 has length of 3

* **Cycle**: Path that starts and ends at the same vertex

E.g.: 0 -> 1-> 4 -> 0 is a cycle

## Types

|  |  |
| --- | --- |
| Undirected Graph | Edge (u, v) implies edge (v, u) |
| Directed Graph | Edges are unidirectional.   * **Directed Cyclic Graph**: A directed graph which contains at least one cycle. * **Directed Acyclic Graph**: A directed graph which contains no cycle. |
| Weighted Graph | Each edge is not treated equally, but some edges have a larger weight than others. |
| Trees | Trees are a subset of graph. Check the [Tree session](#_Tree) for me details. |

## Representations

|  |  |
| --- | --- |
| Edge Set | Commonly used. |
| Adjacency List | Commonly used, especially for space graph (= big graph). |
| Adjacency Matrix | It’s complex, so not commonly used. |

## Applications

Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.

Graphs are used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale, etc.

Shortest path in maps

## Traversal

### Breadth First Search (BFS)

**Problem**

A BFS implementation puts each vertex of the graph into one of two categories:

* Visited
* Not visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

**Algorithm**

The algorithm works as follows:

1. Put a **starting vertex** (any) at the **back** of a Queue.
2. Take the **front** item of the Queue and add it to the Visited List.
3. Find all of that vertex's adjacent nodes. And add those aren't in the Visited List to the **back** of the Queue.
4. Keep repeating steps 2 and 3 until the Queue is empty.

Note: The graph might have two different disconnected parts, so to make sure that we cover every vertex, we can also run BFS on every node.

**Example**

|  |  |
| --- | --- |
| We create an undirected graph with 5 vertices. | undirected graph with 5 vertices |
| We choose vertex 0 as starting point. BFS puts it in the Visited List and puts all its adjacent vertices in the Queue. | visit start vertex and add its adjacent vertices to queue |
| Next, we visit the element at the front of queue – which is 1, so we put 1 to the Visited List.  Then go to its adjacent nodes. Its adjacent vertices are 0 and 2. Because 0 has already been visited and 2 has already in the Queue, we keep the rest of the Queue unchanged. | visit the first neighbour of start node 0, which is 1 |
| Next, we visit vertex 2, so we put 2 to the Visited List. This vertex has an unvisited adjacent vertex – which is 4, so we put 4 to the back of the Queue. | visit 2 which was added to queue earlier to add its neighbours |
| Next, we visit vertex 3, so we put 3 to the Visited List. This vertex has no unvisited adjacent vertex, so we keep the rest of the Queue unchanged. | visit |
| Only 4 remains in the Queue. We visit it: | visit last remaining item in queue to check if it has unvisited neighbours |
| Because the Queue is empty, we have completed the BFS of the graph. |  |

**Time complexity**: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

**Space complexity**: O(V) as it uses a queue to keep track of the vertices that need to be visited.

### Depth First Search (DFS)

**Problem**

A BFS implementation puts each vertex of the graph into one of two categories:

* Visited
* Not visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

**Algorithm**

The algorithm works as follows:

1. Put a **starting vertex** (any) at the **back** of a Stack.
2. Take the **top** item of the stack and add it to the Visited List.
3. Find all of that vertex's adjacent nodes. And add those aren't in the visited list to the **top** of the Stack.
4. Keep repeating steps 2 and 3 until the Stack is empty.

**Example**

|  |  |
| --- | --- |
| We create an undirected graph with 5 vertices. | We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack. |
| We choose vertex 0 as starting point. DFS puts it in the Visited List and puts all its adjacent vertices in the Stack. |  |
| Next, we visit the element at the top of Stack – which is 1, so we put 1 to the Visited List.  Then go to its adjacent nodes. Its adjacent vertices are 0 and 2. Because 0 has already been visited and 2 has already in the Stack, we keep the rest of the Stack unchanged. |  |
| Next, we visit vertex 2, so we put 2 to the Visited List. This vertex has an unvisited adjacent vertex – which is 4, so we put 4 to the top of the Stack. |  |
| Next, we visit vertex 4, so we put 4 to the Visited List. This vertex has no unvisited adjacent vertex, so we keep the rest of the Stack unchanged. | Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it. |
| Only 3 remains in the queue. We visit it. | After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph. |
| Because the queue is empty, we have completed the BFS of the graph. |  |

**Time complexity**: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

**Space complexity**: O(V + E) as it uses a queue to keep track of the vertices that need to be visited.

#### BFS vs DFS

|  |  |  |
| --- | --- | --- |
|  | **BFS** | **DFS** |
| **Data Structure** | Uses **Queue** data structure for finding the shortest path. It works on the concept of FIFO (First In First Out). | Uses **Stack** data structure. It works on the concept of LIFO (Last In First Out). |
| **Tranversal** | First walk through all nodes on the same level before moving on to the next level.  So builds the tree **level by level**. | Begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.  So builds the tree **sub-tree by sub-tree**.  bfs-vs-dfs-(1) |
| **Suitable for** | More suitable for searching vertices closer to the given source. | More suitable when there are solutions away from source. |
| **Applications** | Bipartite graphs, shortest paths, etc.  If weight of every edge is same, then BFS gives shortest path from source to every other vertex. | Cyclic graphs and finding strongly connected components, etc. |

### Uniform Cost Search (UCS)

## Operations & Complexities

### Breadth First Search

Time complexity: O(V + E), where V is the number of nodes and E is the number of edges.

Sace complexity: O(V).

### Depth First Search

Time complexity: O(V + E), where V is the number of nodes and E is the number of edges.

Sace complexity: O(V).

## C++ Examples

### Presentation

|  |  |  |
| --- | --- | --- |
| Edge Set |  |  |
| Adjacency List | #include <iostream>  #include <vector>  using namespace std;    void addEdge(vector<vector<int>>& adj, int i, int j) {      if (i >= adj.size() || j >= adj.size()) {          cout << "FAIL" << endl;          return;      }      adj[i].emplace\_back(j);      adj[j].emplace\_back(i);     // Undirected  }    void display(vector<vector<int>> adj) {      for (int i = 0; i < adj.size(); i++) {          cout << i << ":";          for (int j = 0; j < adj[i].size(); j++) {              cout << " " << adj[i][j];          }          cout << endl;      }  }    // NOTE: Due to fixed vector size, this code only works when maximum value of all vertices is not greater than number of vertices  int main() {      // Create a graph with 4 vertices and no edges      int V = 4;      vector<vector<int>> adj(V);        // Now add edges one by one      addEdge(adj, 0, 1);      addEdge(adj, 0, 2);      addEdge(adj, 1, 2);      addEdge(adj, 2, 3);        cout << "Adjacency list representation:" << endl;      display(adj);        return 0;  } | Adjacency List Representation:  0: 1 2  1: 0 2  2: 0 1 3  3: 2 |
| Adjacency Matrix |  |  |

### Tranversal

#### BFS

|  |  |
| --- | --- |
|  |  |

#### DFS

|  |  |
| --- | --- |
|  |  |

# Hash Table

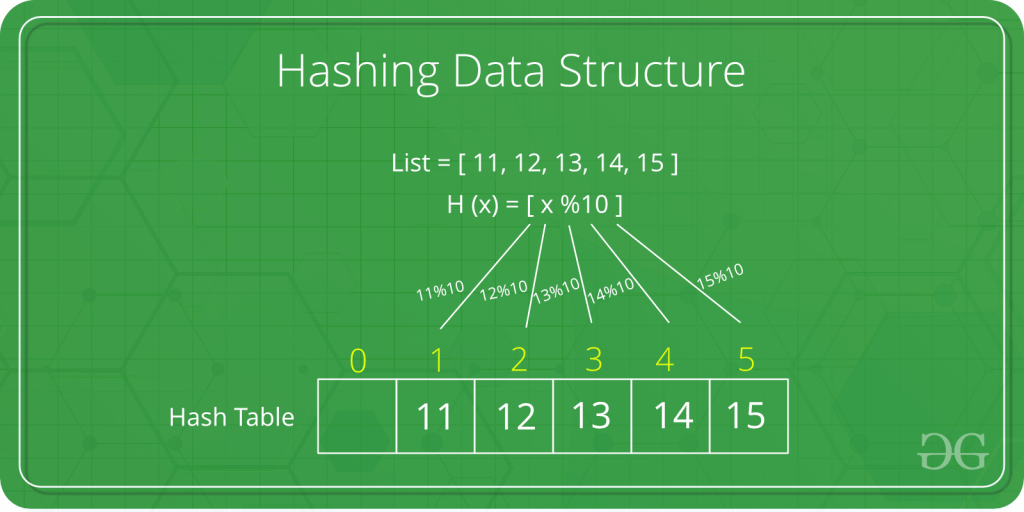
## Definition

Hashing is a popular technique for storing and retrieving data **as fast as possible**. That’s because hasing performs optimal searches.

**What?**

Hashing is a **technique of mapping keys and values into a hash table by using a hash function**. The efficiency of mapping depends on the efficiency of the hash function used.

Example: Hash function H(x) maps the value x at the index x % 10 in a list of [11,12,13,14,15]. The output hast table will store value at positions {1,2,3,4,5} respectively:



**Why?**

For example, in a Balanced Binary Search Tree, the time complexity for searching, inserting and deleting any element is O(logh). What if we want to do the same operations in a faster way? Here hashing comes into play.

In hashing, all the above operations can be performed in **O(1)**. Note thatin worst case, hashing remains O(n), but the average case only takes O(1).

## Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Meaning | Time Complexity | Space Complexity |
| HashTable | Create a new hash table | O(1) | O(1) |
| Delete | Delete a particular key-value pair from the hash table | O(1) | O(1) |
| Get | Search a key and return the value associated with that key | O(1) | O(1) |
| Put | Insert a new key-value pair | O(1) | O(1) |
| DeleteHashTable | Delete the hash table | O(1) | O(1) |

## Components

### Hash Table

Hash table helps store data in a way that it's easy to find. This makes searching of an element very efficient.

### Hash Function

Hash function **converts a given big number to a small integer value** **which is used as an *index* in hash table**.

In other words, a hash function is used to transform a given key into a specific slot index. Its main job is to map each and every possible key into a unique slot index. If every key is mapped into a unique slot index, then the hash function is known as a perfect hash function.

A **good hash function** should have following properties:

* Efficiently computable.
* Uniformly distribute the keys (each table position equally likely for each).
* Minimize collisions.
* Lave a low load factor (number of items in table divided by size of the table).

### Collision Handling

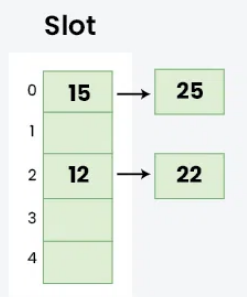
Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a **newly inserted key maps to an already occupied slot in hash table is called collision**.

We must handle collisions. Following are some ways:

#### Chaining

The idea is putting all elements, that hash into the same slot index (or different elements having the same hash value), into a linked list (or dynamic sized array).

Example: A sequence of keys as 12, 22, 15, 25 and a hash function as "key mod 5":



**Pros:**

* Simple to implement.
* Hash table never fills up, we can always add more elements to the chain.
* Less sensitive to the hash function or load factors.
* It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

**Cons:**

* The cache performance of chaining is not good as keys are stored using a linked list. But we can improve it by using a dynamic sized array.
* Wastage of space as some parts of the hash table are never used.
* If the chain becomes long, then search time can become O(n) in the worst case.
* Uses extra space for links.

#### Open Addressing

Here all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

Different ways of Open Addressing:

##### Linear Probing

Here if the slot that we get is already occupied, then we check for the **next slot which is right after the current slot**.

The typical gap between two probes is 1. Let *hash(x)* be the slot index computed using a hash function and *S* be the table size:

* If slot *hash(x) % S* is full, then we try *(hash(x) + 1) % S*
* If slot *(hash(x) + 1) % S* is also full, then we try *(hash(x) + 2) % S*
* If slot *(hash(x) + 2) % S* is also full, then we try *(hash(x) + 3) % S*
* Etc.

Example: A sequence of keys as 50, 70, 76, 85, 93 and a hash function as "key mod 5":

|  |  |
| --- | --- |
| * Key is 50 which is mapped to slot 0 (50 % 5 = 0). * Key is 70 which is mapped to slot 0 (70 % 5 = 0). But slot 0 is already occupied, the next empty slot which is slot 1 is used. * Key is 76 which is mapped to slot 1 (76 % 5 = 1). But slot 1 is already occupied, the next empty slot which is slot 2 is used. * Key is 85 which is mapped to slot 1 (85 % 5 = 0). But slot 0 is already occupied, the next empty slot which is slot 3 is used. * Key is 93 which is mapped to slot 1 (93 % 5 = 3). But slot 3 is already occupied, the next empty slot which is slot 4 is used. |  |

##### Quadratic Probing

Here we look for the i2'th slot in the ith iteration. If the slot that we get is already occupied, then we check for the **next slot which is proportionally to the current slot**.

Let *hash(x)* be the slot index computed using hash function and *S* be the table size.

* If slot *hash(x) % S* is full, then we try *(hash(x) + 1\*1) % S*
* If slot *(hash(x) + 11) % S* is also full, then we try *(hash(x) + 22) % S*
* If slot *(hash(x) + 22) % S* is also full, then we try *(hash(x) + 33) % S*
* Etc.

Example: Insert keys 22, 30, and 50 into a hash table of size 7, where hash function is h(x) = x % 7 and collision resolution strategy is f(i) = i2.

|  |  |
| --- | --- |
| * Key is 22 which is mapped to slot 1 (22 % 7 = 1). * Key is 30 which is mapped to slot 2 (30 % 7 = 2). * Key is 50 which is mapped to slot 1 (50 % 7 = 1). But slot 1 is already occupied, so we'll search for slot (1 + 12 = 2) % 7 = 2. But slot 2 is also occupied, so we'll search for slot (1 + 22) % 7 = 5. |  |

##### Double Hashing

Here the increments for the probing sequence are computed by **using another hash function**. In particular, we use another hash function *hash2(x)* and look for the *i\*hash2(x)* slot in the ith rotation.

Let *hash(x)* be the slot index computed using hash function and *S* be the table size.

* If slot *hash(x) % S* is full, then we try *(hash(x) + 1\*hash2(x)) % S*
* If slot *(hash(x) + 1\*hash2(x)) % S* is also full, then we try *(hash(x) + 2\*hash2(x)) % S*
* If slot *(hash(x) + 2\*hash2(x)) % S* is also full, then we try *(hash(x) + 3\*hash2(x)) % S*

Example: Insert keys 27, 43, 692, 72 into the hash table of size 7, where first hash-function is h1​(k) = k % 7 and second hash-function is h2(k) = 1 + (k % 5)

|  |  |
| --- | --- |
| * Key is 27 which is mapped to slot 6 (22 % 7 = 6). * Key is 43 which is mapped to slot 1 (43 % 7 = 1). * Key is 692 which is mapped to slot 6 (692 % 7 = 6). But slot 6 is already occupied, so we'll search for slot (6 + 1 \* (1 + (692 % 5))) % 7 = 2. * Key is 72 which is mapped to slot 2 (72 % 7 = 2). But slot 2 is already occupied, so we'll search for slot (2 + 1 \* (1 + (72 % 5))) = 5. |  |

**Pros:**

* A slot can be used even if an input doesn’t map to it.
* Better cache performance as everything is stored in the same table.
* No links, so no extra space wastage.

**Cons:**

* Hard to implement because it requires more computation.
* Table may become full.
* Only used when the frequency and number of keys is known.

## Applications

Design a system for storing employee records using phone numbers as keys.

We can think of using the following data structures to maintain information about different phone numbers:

* Array of phone numbers and records.
* Linked List of phone numbers and records.
* Balanced Binary Search Tree with phone numbers as keys.

We want the following queries to be performed efficiently:

* Insert a phone number and corresponding information.
* Search a phone number and fetch the information.
* Delete a phone number and related information.

But none of above data structures are really efficent. So we use hash table.

**Design for hash table:**

* A hash table stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.
* A hash function converts a big phone number to a small integer value which is used as index in the hash table. For phone numbers, a bad hash function is to take first three digits. A better function is consider last three digits.